Представление целых чисел

- https://github.com/blackav/hse-caos-2019/blob/master/00-lectures/02-integers/lecture02.pdf

Байты и октеты

- Обычно, мы мы говорим байт, и имеем в виду под этим 8 бит, но это не так
- В стандарте C++ байт это `char`, минимально адресуемая единица памяти
- Количество бит в `char` определяется константой `CHAR_BIT`, но `CHAR_BIT` в стандарте `>= 8`.
- А есть слово "октет" - это всегда 8 бит.

Sizeof operator

```
sizeof( type )
sizeof expression
```

`sizeof` - возвращает размер типа или объекта

sizeof.cpp

Целые типы с разным количеством битов в зависимости от окружения

- `short` - как минимум 16 бит
- `int` - как минимум 16 бит
- `long` - как минимум 32 бита
- `long long` - как минимум 64 бита

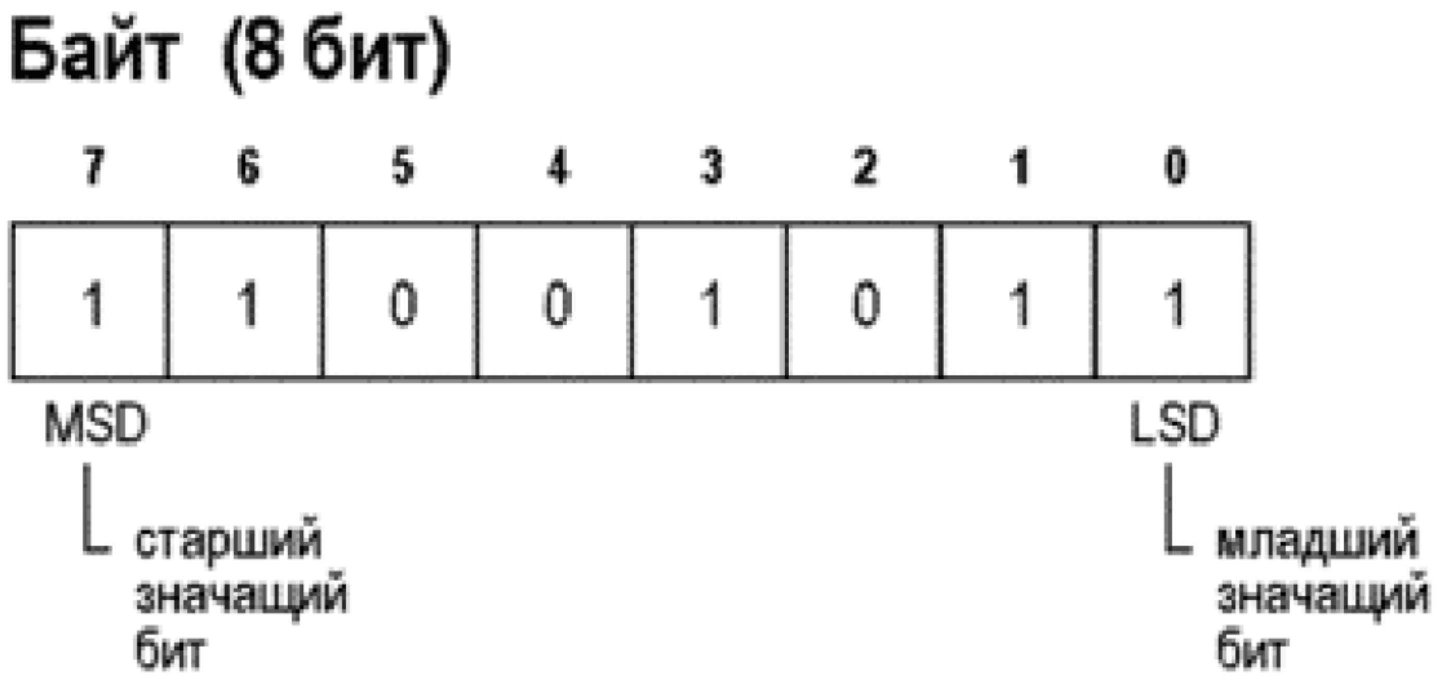
`signed`, `unsigned` - знаковый или беззнаковый тип. Если у вышеперечисленных не указывать явно эти ключевые слова, то будет `signed`.

Целые типы, которые лучше использовать

- `cstdint` - `intN_t`, `uintN_t`.
- `cinttypes` - I/O макросы.

cinttypes.cpp

Разряды

- 
Байт (8 бит)
7 6 5 4 3 2 1 0
1 1 0 0 1 0 1 1
MSD
старший значащий бит
LSD
младший значащий бит

Знаковые целые числа

- Надо где-то хранить знак числа
- Стандарт C++ допускает три типа представления чисел:
    - Прямой код
    - Обратный код
    - Дополнительный код

Прямой код

Положительное	Двоичное	Прямой код
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	5
7	0111	7
8	1000	-0
9	1001	-1
10	1010	-2
11	1011	-3
12	1100	-4
13	1101	-5
14	1110	-6
15	1111	-7

Обратный код

Положительное	Двоичное	Прямой код
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	5
7	0111	7
8	1000	-7
9	1001	-6
10	1010	-5
11	1011	-4
12	1100	-3
13	1101	-2
14	1110	-1
15	1111	-0

Дополнительный код

Положительное	Двоичное	Прямой код
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	5
7	0111	7
8	1000	-8
9	1001	-7
10	1010	-6
11	1011	-5
12	1100	-4
13	1101	-3
14	1110	-2
15	1111	-1

Дополнительный код

- $-x = \sim x + 1$
- $\left[-2^{n-1}, 2^{n-1}\right)$ - диапазон, где n - количество байт в типе
- $-0_{10} = 0_{10}$, докажем для 4 битного типа
  $-0_{10} = \ 0_{10} + 1 = 1111_2 + 1_2 = 0000_2 = 0$
- Операции сложения и вычитания n-битных беззнаковых чисел дают правильный результат для знаковый чисел.

Дополнительный код

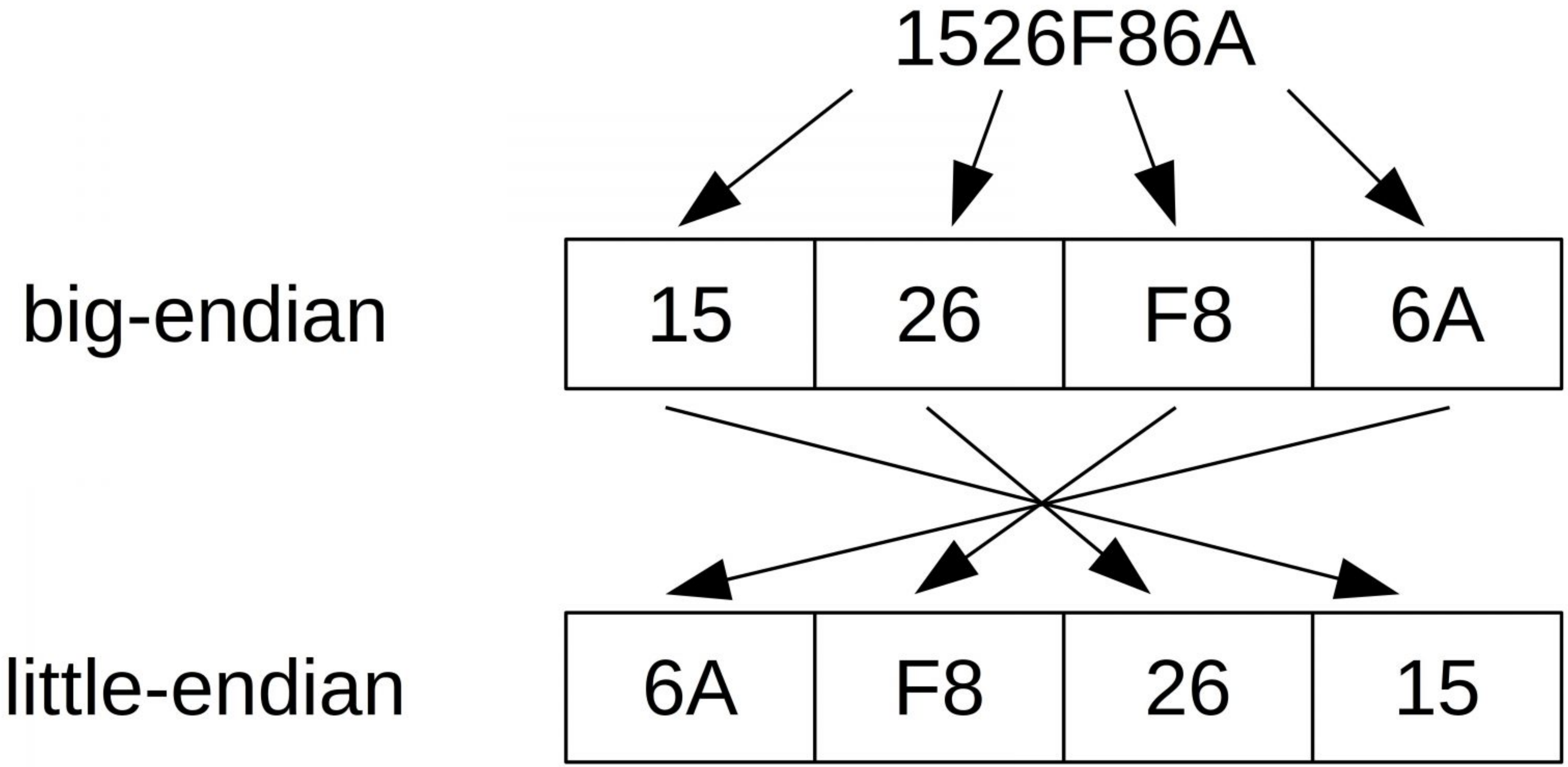
- Переполнение знакового числа это undefined behavior
- `-fwrapv` – выполнять знаковые операции по модулю 2^n, но компилятор не делает никаких оптимизационных предположений
- `-fsanitize=undefined` – проверка на undefined behavior при работе программы

overflow.cpp

Еще undefined behavior при знаковых операциях

- Сдвиг на число бит, большее размера типа – undefined behavior
- Сдвиг отрицательного числа влево – undefined behavior
- Сдвиг на отрицательную величину – undefined behavior

Порядок байт



endian.cpp

Char

- `char` - может быть либо знаковым, либо беззнаковым (implementation defined behavior)
- Можно переключаться между режимами, через `-fsigned-char`, `-funsigned-char`. (точно работает в `clang++`, `g++`)

char.cpp

Максимальные и минимальные значения типов

- C++ style
- `#include <limits>`
- `std::numeric_limits<type>`
- C style
- `#include <climits>`
- `CHAR_BIT, MB_LEN_MAX, CHAR_MIN, CHAR_MAX, SCHAR_MIN, SHRT_MIN, INT_MIN, LONG_MIN, LLONG_MIN, SCHAR_MAX, SHRT_MAX, INT_MAX, LONG_MAX, LLONG_MAX, UCHAR_MAX, USHRT_MAX, UINT_MAX, ULONG_MAX, ULLONG_MAX, PTRDIFF_MIN, PTRDIFF_MAX, SIZE_MAX, SIG_ATOMIC_MIN, SIG_ATOMIC_MAX, WCHAR_MIN, WCHAR_MAX, WINT_MIN, WINT_MAX`

Каст целых чисел

- Преобразование знакового типа в беззнаковый тип сохраняет битовое представление
- Преобразование более широкого типа в более узкий тип отсекает старшие биты

cast.cpp

Представление вещественных чисел

- https://habr.com/ru/articles/745640/
- http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/04-float.pdf

Представление вещественных чисел

Для сохранения чисел с плавающей точкой процессор компьютера выполняет следующий алгоритм:

- Переводит число из десятичной в двоичную систему
- Получившиеся число переводит в экспоненциальную запись
- Число в экспоненциальной записи поместить в 32 бита (для примера будем брать регистр объемом памяти равный 32 битам).
  - `float` - 32 бита
  - `double` - 64 бита

Пример представления вещественных чисел. Перевод в двоичную систему счисления

Возьмем числа 0.59375, -8.75, 3.9 и попробуем перевести их в двоичный вид.

- $-8.75_{10} = -1000.11_2$
- $0.59375_{10} = 0.10011_2$
- $3.9_{10} = 11.11100(1100)_2$

Пример представления вещественных чисел. Перевод в экспоненциальную запись

Возьмем числа 0.59375, -8.75, 3.9 и попробуем перевести их в двоичный вид.

- $-8.75_{10} = -1.00011_2 * 10_2^3$
- $0.59375_{10} = 1.0011_2 * 10_2^{-1}$
- $3.9_{10} = 1.111100(1100)_2 * 10_2^1$

Пример представления вещественных чисел. Типы

- Single precision: 32 bits

s	exp	frac
1	8-bits	23-bits

- Double precision: 64 bits

s	exp	frac
1	11-bits	52-bits

- Extended precision: 80 bits (Intel only)

s	exp	frac
1	15-bits	63 or 64-bits

Пример представления вещественных чисел. Запись в 32 бита

S	E								F																						
1																															

$-8.75_{10} = -1.00011_2 * 10^3$

S	E								F																						
0																															

$0.59375_{10} = 1.0011_2 * 10^{-1}$

S	E								F																						
0																															

$3.9_{10} = 1.111100(1100)_2 * 10^1$

Пример представления вещественных чисел. Запись мантисы

S	E								F																						
1									0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$-8.75_{10} = -1.00011_2 * 10^3$

S	E								F																						
0									0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$0.59375_{10} = 1.0011_2 * 10^{-1}$

S	E								F																						
0									1	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1

$3.9_{10} = 1.111100(1100)_2 * 10^1$

Пример представления вещественных чисел. Неправильная запись экспоненты

S	E								F																						
1	1	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$-0.75_{10} = 1.1_2 * 10^{-1}$

S	E								F																						
0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$4_{10} = 1.0_2 * 10^{2}$

И тут сразу можно заметить две проблемы:

- процессор при сравнении двух чисел будет побитово проходиться по ним и сравнивать значения. Дойдя до числа с отрицательным битом, он увидит там единицу и решит, что это число больше, и операция сравнения будет неправильной
- мы выделяем один бит на хранения знака из-за чего наша и точность станет ниже.

Пример представления вещественных чисел. Запись экспоненты

S	E								F																						
1	1	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$-8.75_{10} = -1.00011_2 * 10^3$

S	E								F																						
0	0	1	1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$0.59375_{10} = 1.0011_2 * 10^{-1}$

S	E								F																						
0	1	0	0	0	0	0	0	0	1	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1

$3.9_{10} = 1.111100(1100)_2 * 10^1$

диапазон чисел от 0 до 255 был разделен на две части, при этом число посередине $(12710 = 01111111_2)$ стало восприниматься как 0. Все числа больше 127 воспринимались процессором как положительные, а меньше 127 - как отрицательные.

Специальные значения

S	E								F																						
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

-Infinity

S	E								F																						
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

+Infinity

S	E								F
-	1	1	1	1	1	1	1	1	-

NaN